Problem 1

Discription

A modification could be adding a new array $S[n]$ for the number of shortest paths from s to vertex n, and an array $D[n]$ for length of the shortest path from s to vertex n. Obviously $S[n]$ is our answer. We initially set

$$\begin{aligned} S[s] &= 1 \\ S[n \neq s] &= 0 \\ D[s] &= 0 \\ D[n \neq s] &= \infty \end{aligned}$$

In each iteration of original BFS, we will extract u from the queue, and find all its neighbours v. We will discard the test of whether v has been visited or not, since it is embedded in the following test. We will compare $D[u]+1$ and $D[v]$. If $D[u]+1 < D[v]$, we will do $D[v] \leftarrow D[u]+1, S[v] = 1$ and put v into queue. If $D[u]+1 = D[v]$, then we will do $S[v] \leftarrow S[v]+1$, however, we will not put v into queue. If $D[u]+1 > D[v]$, then we do nothing.

Pseudocode

Algorithm 1 Modified BFS

```
1:  Initialize $D$ and $S$ as stated before
2:  Initialize an empty queue $Q$
3:  ENQUEUE $(Q, s)$
4:  while $Q \neq \phi$ do
5:      $u \leftarrow$ DEQUEUE $(Q)$
6:      for $v \in Adj[u]$ do
7:          if $D[u] + 1 < D[v]$ then                ▷ Need to add $v$ into queue
8:              $D[v] \leftarrow D[u] + 1$
9:              $S[v] \leftarrow 1$
10:             ENQUEUE $(Q, v)$
11:         else if $D[u] + 1 = D[v]$ then           ▷ Another path to $v$ exists
12:             $S[v] \leftarrow S[v] + 1$
    return $S$
```

Problem 2

Discription

During each iteration of topological sorting, there are possibly many vertices with in-degree zero. The original algorithm gives not specification on which to select. We claim that by greedily selecting the smallest available index at each itereation respectively, one can achieve a minimum lexicographic ordering at the end. The modification will be replacing the queue in topological sorting with a min-heap.

A min-heap insertion and extraction requires $O(\log |V|)$ time, therefore the total time complexty will be $O((|V| + |E|) \log |V|)$.

Pseudocode

Algorithm 2 Modified topological sorting

```
1: Initialize H to be an empty heap
2: for u in V do
3:     if in-degree[u] = 0 then
4:         add u to H
5: while H is non-empty do
6:     u = H.top, H.pop
7:     Outpur u
8:     for v in Adj[u] do
9:         in-degree[v] = in-degree[v]-1
10:        if in-degree[v] = 0 then
11:            add v to H
```

Problem 3

Proof. The only terminating exit of the algorithm is the step four. The termination condition is that the maximum connected component C_k of $T - \{v\}$ have size not greater than $|V|/2$. It means that every connected components of $T - \{v\}$ have size not greater thant $|V|/2$. Therefore, when the algorithm terminates, it yields the centroid.

Now we will prove that the algorithm will terminate. We prove it by contradiction. Suppose it does not terminate. Let $\{v_n\} = v_1, v_2, \dots$ be an infinite sequence of $v's$ generated in the step 5 of the algorithm. However, the total amount of vertices on the tree is finite, which indicates that there are repetitions in $\{v_n\}$. Without lost of generality, let $s = v_i = v_j$ for an $i < j$.

We have $i \neq j - 1$. This is obvious since the algorithm forces to find a neighbor of current poing v, thus any adjacent points in $\{v_n\}$ are not the same. Let $t = v_{i+1}$. Split the tree by removing the edge (s, t), we get two connected components: C_s, C_t. Because there is no loop on a tree, s and t are disconnected. Thus to get back from t to s, we can only take the edge (t, s) again latter in the sequence. The movement from s to t indicates $|C_t| > |V|/2$, and the movement from t to s indicates $|C_s| > |V|/2$. Therefore, $|C_s| + |C_t| > |V|$, but $|C_s| + |C_t| = |V|$ obviously and this incurs contradiction. □

Problem 4

From a grid M of size $n \times n$, we generate a graph with $2n + 2$ points as following: $s, t, a_1, \dots, a_n, b_1, \dots, b_n$.

Generate Graph

Algorithm 3 Generation of Graph

```
1: for $i$ from 1 to $n$ do
2:     addedge $s, a_i, 1$        ▷ addedge $x, y, d$: add a directed edge from $x$ to $y$ of with capacity $d$
3:     addedge $b_i, t, 1$
4: for $i$ from 1 to $n$ do
5:     for $j$ from 1 to $n$ do
6:         if $M[i][j]$ is white then
7:             addedge $a_i, b_j, 1$
```

Then we run a maxflow algorithm to get the maximum flow k from s to t. We will also record E, the set of edges that has been used in the maxflow between $a's$ and $b's$. Notice that every capacity in the graph is an integer, so after only performing addition and substraction operations in an maxflow algorithm, the flow in every edge must also be integral. Hence a used edge in E means its flow is exactly one. For each edge (a_i, b_j) in E, we will place the token on M with axis (i, j), row $= i$, col $= j$.

Lemma 1. *Every token is placed on a white cell.*

This is obvious since E is a subset of all edges between $a's$ and $b's$, which is generated if and only if the corresponding position is white.

Lemma 2. *Each row and each column of the grid has* ***at most*** *one token.*

Suppose there are $t > 1$ tokens in row i, then it there are t edges from a_i is used, which means that t units of flow have passed a_i. This is not possible since the maximum in-flow of a_i is 1, thus flow balance is broken. The case of $t > 1$ tokens in col j can be proved likewise.

Lemma 3. $k = |E| \leq n$

E is a subset of all edges between $a's$ and $b's$. Since all edges between $a's$ and $b's$ are all in one direction from $a's$ to $b's$, Then

$$k = f(S, T) = \sum_{e \in E} 1 = |E|$$

Where $S = \{s, a_1, \dots, a_n\}, T = \{b_1, \dots, b_n, t\}$.

Since the out-degree of s is n, the maximum flow $k \leq n$.

Lemma 4. *Each row and each column of the grid has exactly one token if and only if $k = n$.*

If $|E| = k = n$. Let x_i be the number of tokens in row i. Since the total amount of tokens is $|E| = n$ by defintion, $\sum_{i=1}^{n} x_i = n$. By lemma 2, $x_i \leq 1$, thus $x_i = 1$. The columns can be proved likewise.

If each row and each column has exactly one token, then obviously $|E| = n$. By lemma 3, $k = n$.

Using Lemma 4, we can check that the placement stated before is valid. Hence the original problem can be reduced to a maxflow problem with polynomial running time. If $k = n$, then we have a valid placement; if $k < n$, then no such placement exists.

Problem 5

To find vertices that forms a remote pair with u, we could simply run a bfs starting with u to get the shortest distance from u in D. And then we check each $D[v]$ one by one to see if $D[v] > \frac{n}{2}$. This algorithm is correct since BFS can return shortest paths of an unweighted undirected graph, and the checking part is by definition of the remote pair. The running time for BFS is $O(n + m)$, and the checking process requires $O(n)$, so the total running time is $O(n + m)$.

Lemma 5. *If u, v is a remote pair, it is also a tenuous pair.*

Proof. Consider a BFS tree generated from u. We claim that there exists a layer l, $1 \leq l \leq \lfloor \frac{n}{2} \rfloor$, such that this layer contains only one vertex. In another word, $D[x] = l$ has only one solution for x. Suppose not the case, then all the layers between 1 and $\lfloor \frac{n}{2} \rfloor$ have at least 2 vertices, altogether we have $\lfloor \frac{n}{2} \rfloor \times 2 \geq n - 1$ vertices. Adding u and v, we have at least $n+1$ vertices, contradiction. Thus let x be the only solution of $D[x] = l$. By properties of BFS tree, all vetices layers less than l connect to layers larger than l via x, otherwise there will be more vertices in layer l. In conclusion, removing x will result in removing all paths from layers less than l to layers larger than l. Since $d(u, v) > \frac{n}{2}$, $v's$ layer is larger than l and thus u, v is also a tenuous pair. $\square$

According to lemma 5, the algorithm described before can be directly used to find remote tenuous pairs. The proof of lemma 5 also gives us a method to find x: after performing the algorithm above, we count vertices of each layers: scan through D and add to counter $counter[D[i]] = counter[D[i]] + 1$ at every iteration. By lemma 5, if there exists a remote pair, then there must exist x such that $counter[D[x]] = 1$, scan and find it, and thus we claim that x is the vertex we are looking after for all tenuous vertices. If there are no remote pairs, then nothing is needed to do.

The running time is $O(n + m)$ plus two scans through D, hence $O(n + m)$ in total.